

❖❖❖

SO YOU WANT TO BE A

WIZARD

by Julia Evans



Here's how I approach
learning hard things
and getting better
at programming!

about this zine

Hi! I'm Julia.



JULIA EVANS
@b0rk
blog: jvns.ca

I don't always feel like a wizard. I'm not the most experienced member on my team. Like most people I find my work difficult sometimes, and I have a TON TO LEARN.

But over the past 5 years, I've learned a few things that have helped me. We'll talk about:

- how asking dumb questions is actually a superpower
- debugging tools that help you FEEL like a wizard
- how learning to write a design doc has helped me
- how to approach learning a complex system
- reading the source code to your dependencies and why that's useful

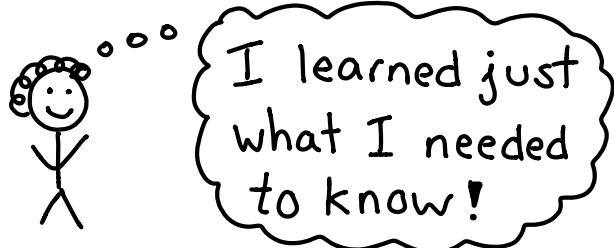
This zine definitely won't teach you to be a wizard by itself, but hopefully it has one or two useful tips!

A lot of it is what I wish I'd known a little earlier in my career !!

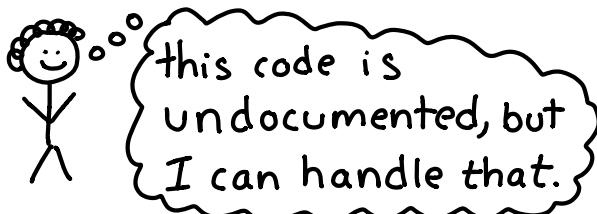
TABLE OF CONTENTS

Here's what we'll cover!

- asking good questions



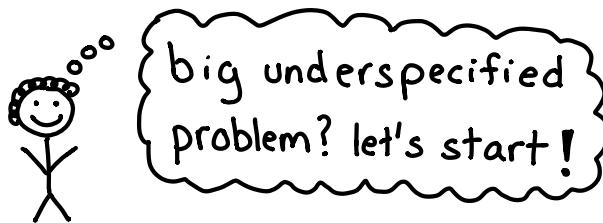
- reading the source code



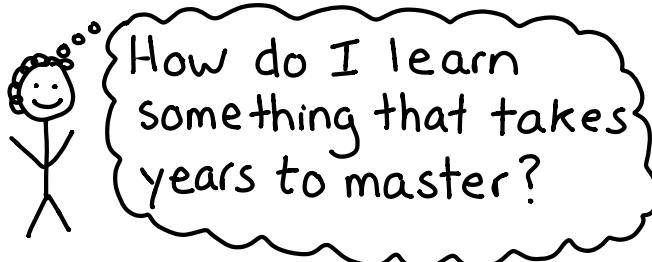
- debugging



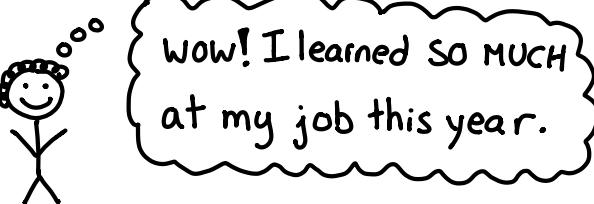
- designing



- building expertise



- strategies for learning



How to be a Wizard Programmer

who can do anything (takes a very long time)

① ASK QUESTIONS. As long as there are people around you who know things you don't, ask them how to do things. Dumb questions. Scary-to-ask questions. Your questions will get less dumb fast.

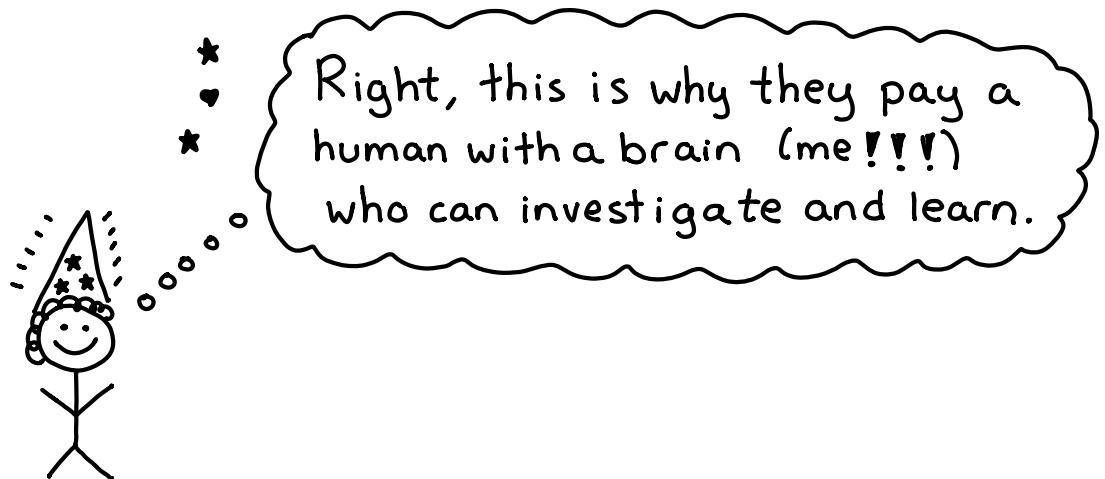
② Run into a problem your coworkers don't know how to solve either.

③ DECIDE YOU WILL FIGURE OUT HOW TO SOLVE THE PROBLEM ANYWAY.
(this is very hard but sometimes it works !!)

The more programming I do, the more issues I run into where:

- I don't know
- my colleagues don't know
- Google doesn't know
- we gotta figure it out anyway

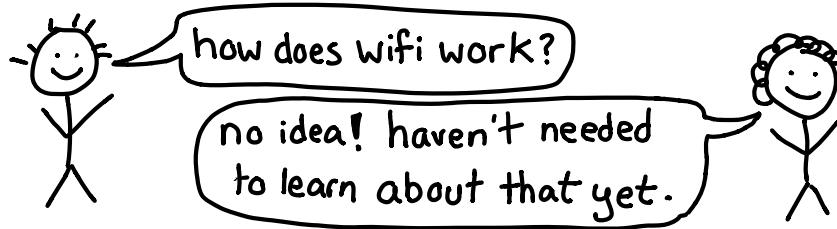
When this happens, I think:



This zine is about what the skill of "figure it out anyway" looks like.

When to invest in understanding?

We work with a lot of abstractions. You don't always need to spend time understanding how they all work under the hood.



But a huge part of becoming a wizard is understanding how a seemingly magical computer system works.

When is it useful to spend time learning how a thing works?

① When you're trying to debug a tricky problem

- Sometimes the libraries you depend on have bugs.
- Often, libraries/systems (like CSS or Linux) have complex abstractions (like the box model or epoll) that take time to learn.

② When you're trying to push the limits / optimize performance

For example, if you're writing data to a file you're always limited by the speed of your disks! Thinking about hardware can be important.

③ When you're trying to innovate

If you're building a new abstraction (like an async library), you gotta understand how the next layer down works! (epoll, select, etc.)

Asking good questions

One of my favorite tools for learning is asking questions of all the awesome people I know!

≡ what's a good question? ≡

good questions:

- ★ are easy for the person to answer
- ★ get you the information you're looking for

Here are some strategies for asking them:

state what you know



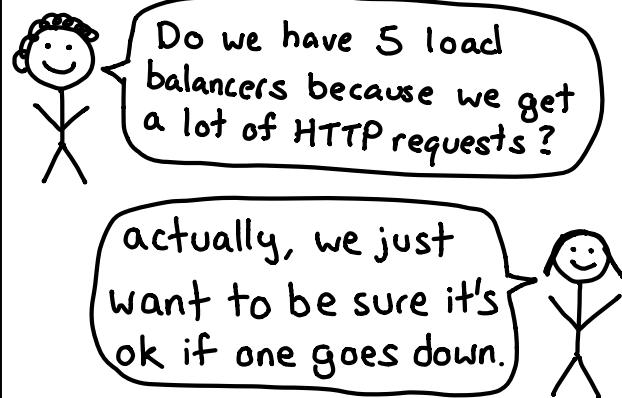
This helps because:

- I'm forced to think about what I know
- I'm less likely to get answers that are too basic or too advanced

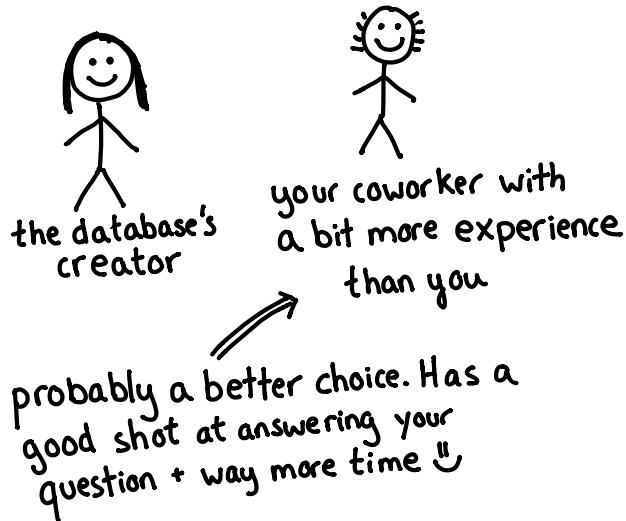
Guessing the answer:

- makes me think!
- helps my coworker see what kind of answer I'm looking for

guess what the answer might be



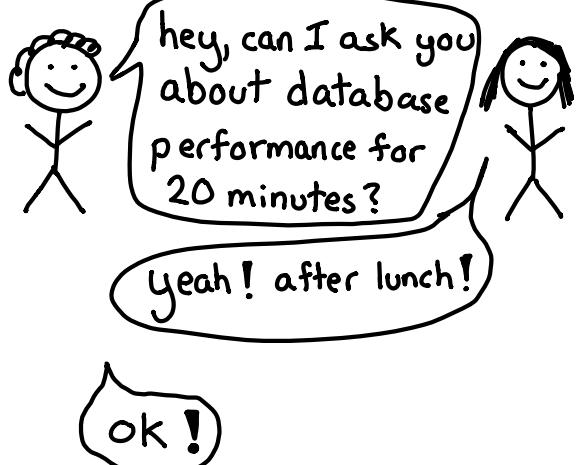
choose who to ask



The person who knows the MOST isn't always the best person to ask!

Often, someone who learned more recently will remember better what it was like to not understand.

find a good time



Especially if I have LOTS of questions, it's good to be respectful of their time !!

do some research



If I spend some time doing research first, I can ask a WAY BETTER question !!

ask yes/no questions



I ❤ asking yes/no questions like this because they're easier to answer and it means I have to focus the question carefully.

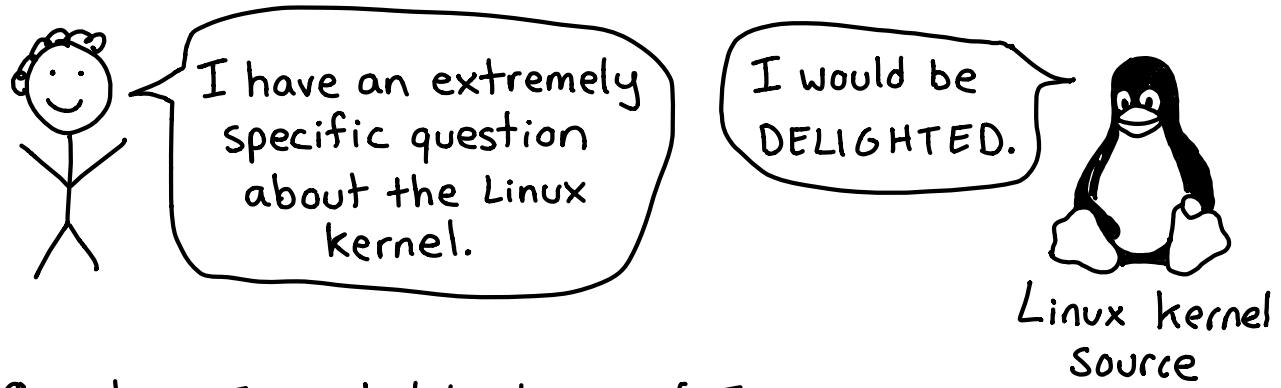
read the source code

Okay, but you can't **ALWAYS** ask people questions!

Sometimes:

- there's no documentation
- your coworkers are busy
- or they don't know the answer
- or you want to know A LOT more details than it is really reasonable to ask about

Luckily, we have open source!!!



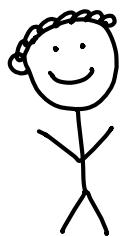
One day, I wanted to know if I could configure a socket on Linux to not queue connections. I googled and got some conflicting answers. But one of the Stack Overflow answers linked directly to the KERNEL CODE!

It looked basically like:

backlog = max(backlog, 8)
 ↑ hardcoded constant!

So it's impossible to set the backlog to 0.
It'll always end up being at least 8 !!

tips for reading code



What does this code do?



not sure, but nobody's touched it for a year and today it's broken.

Ok, time to find out!

Here are some things I've found helpful when dealing with unfamiliar code:

grep for the error message



oh, THAT's what that error message means!

When I see an error message I don't understand, searching the source for it is really easy and sometimes helps.

If the code I'm using is less than a few thousand lines, I like to quickly try to read it all to learn the basics of how it works.

just read all the code



oh, this just runs the "Sync All" function every 10 seconds.

edit the code



what happens if I change THIS?

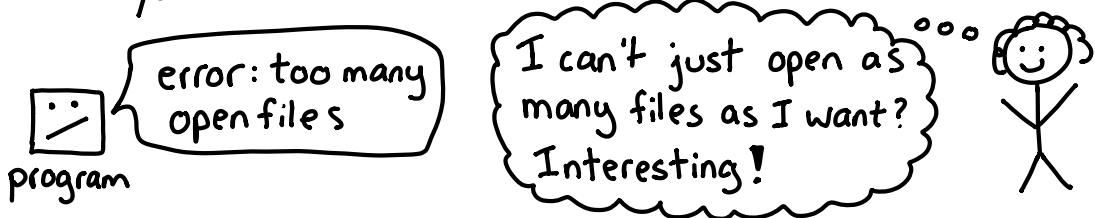
Get your hands dirty!

- step through with a debugger!
- add tests!
- add print statements!
- introduce bugs!
- experiment!
- don't always trust the comments :)

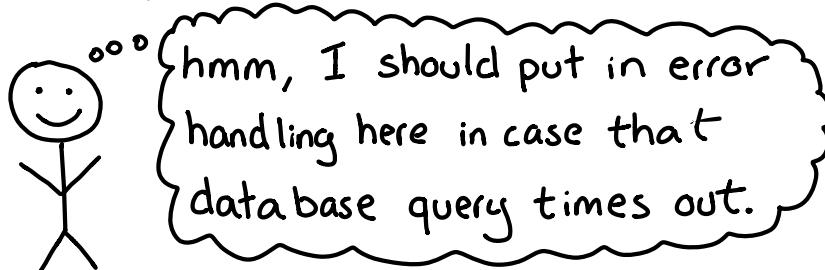
debugging: ❤️ love your bugs ❤️

(thanks to Allison Kaptur for teaching me this attitude!
she has a great talk called "Love Your Bugs".)

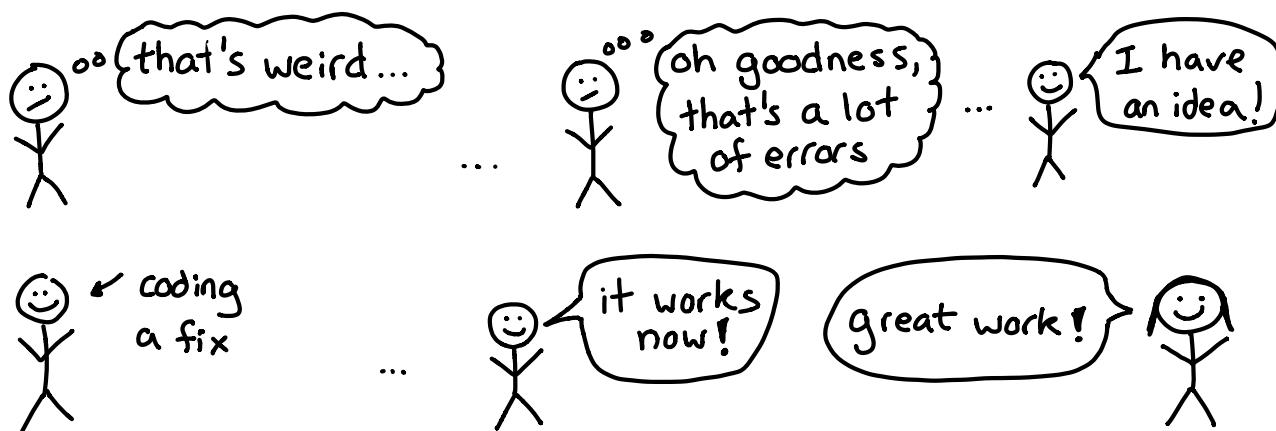
Debugging is a great way to learn. First, the harsh reality of bugs in your code is a good way to reveal problems with your mental model.



Fixing bugs is also a good way to learn to write more reliable code!



Also, you get to solve a mystery and get immediate feedback about whether you were right or not.



Nobody writes great code without writing + fixing lots of bugs. So let's talk about debugging skills a bit!

how I got better at debugging

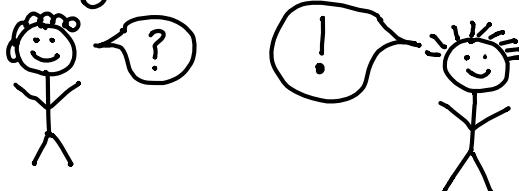
Remember: the bug is happening for a logical reason.

It's never magic. Really. Even when it makes no sense.

Be confident I can fix it

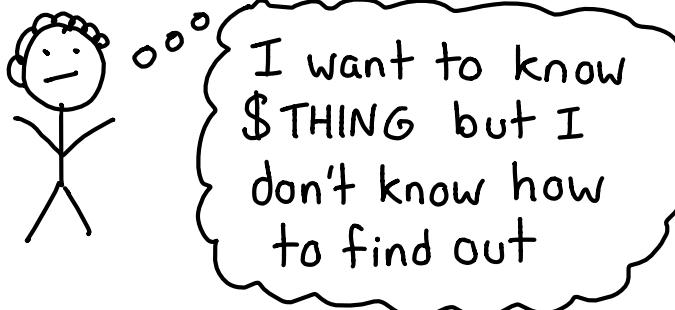


Talk to my coworkers

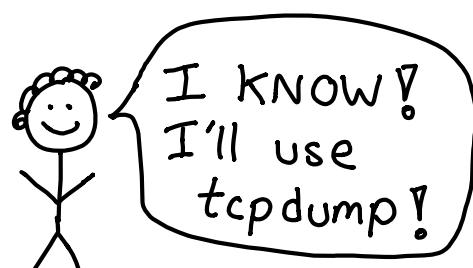


know my debugging toolkit

before:



now:



most importantly: I learned to like it



learning at work

Almost everything I spend time on day to day is something I've learned on the job.



hmm, I need to use Kubernetes at work, but I don't understand it well.

- set aside work time to:
- =>
- read source code
 - ask questions
 - watch talks
 - read docs/blog posts
 - do experiments

Debugging is one way to learn at work. Here are more ways!

follow up on bugs you couldn't figure out



ooh, someone else fixed that. Let's see how!

pay attention to others' code



this PR has some great ideas!

volunteer to do work that seems hard



I'm sure* I'll figure it out.

* I'm not always 100% sure, but it's worth trying!

watch more senior people operate

"that person does AWESOME work. How do they do it?"

don't: advocate for using something at work just because you want to learn it.

learning on my own

go to a conference

especially in an area
I don't know well
(like Linux Kernel networking)

implement something
that seems hard

gzip! tcp! keyboard driver!
debugger!

try a new tool

hmm, can I debug
Python with gdb?

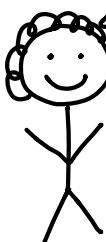


b-trees! epoll! asyncio!

read a paper

Adrian Colyer's
"The Morning Paper"
has amazing paper
summaries

do some experiments



how many
requests /sec
can I serve
with Flask?

* teach /blog it! *

A huge part of my learning process is teaching as I learn!
Reasons it helps:

→ revisiting basic questions
is important.

→ it forces me to realize
when I don't actually
understand something
well yet.

How *does* asynchronous
programming work?

Wait, I didn't realize
Unix groups did that!

learning to ~~design~~ software

It's surprisingly easy to end up in this situation:



A little bit of \heartsuit planning \heartsuit helps me make sure my hard work doesn't go to waste.

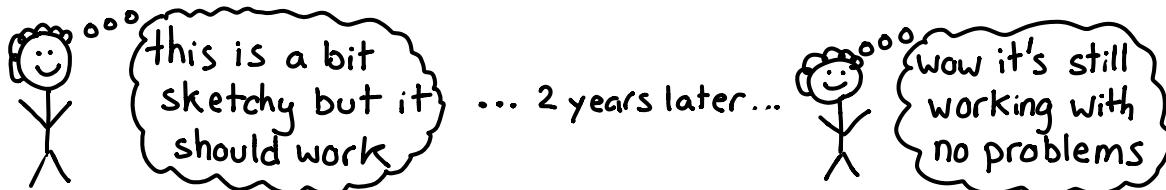
Here are a few things I find helpful to remember:

- ★ you can't predict how requirements will change



I just try my best and deal with changes when they come.

- ★ "good enough" is often really awesome

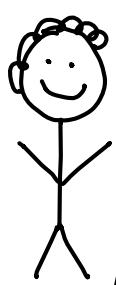


- ★ making a proof of concept can really help.



scenes from writing design docs

when I start writing it



hmm, I hadn't thought about how that part should work before I wrote it down!

people who understand the project better

* me!

* my team!

* my manager!

* other teams!



designing small projects:
still useful

① spend 30 minutes writing

②



this look right?



yep, just one small correction!

when people disagree
(and it goes well)



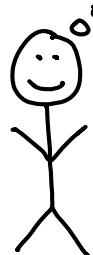
I don't think this is quite right...

let's talk!



we figure out a better plan together!

When I start coding



huh, this is a lot easier now!

3 months into the project

Original plan



what actually happened



designs always change !!

let's build expertise !

Let's zoom out a bit. A lot of the people I admire the most have been working on getting better at what they do for ★ years ★.

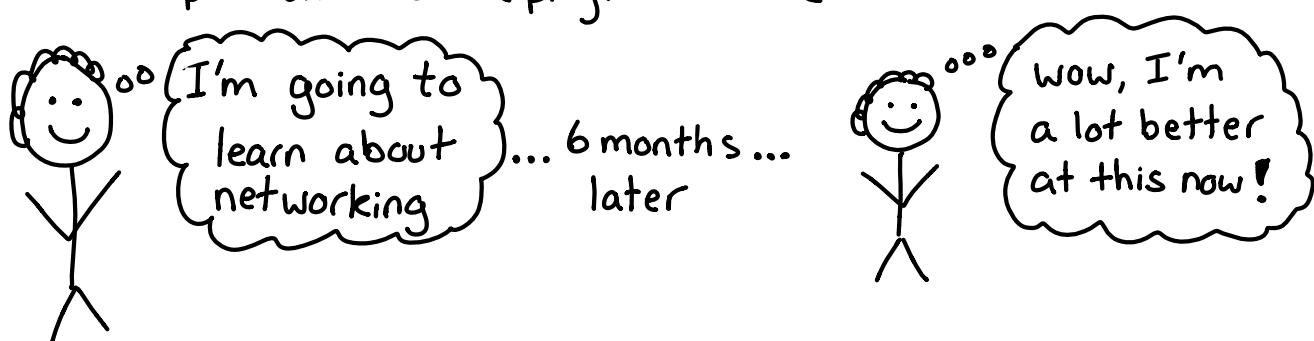
I've found it useful to pick a few things I'm really interested in (like Linux!) and focus on those.

Things I've spent significant amounts of time (at least a year) working on getting better at:

- Linux networking !
- debugging + profiling tools !
- machine learning !
- planning projects at work !
- technical writing / speaking !

There are lots of things (Go! Databases! Javascript!) that are important and I know a little about but haven't spent that much time on. That's okay!

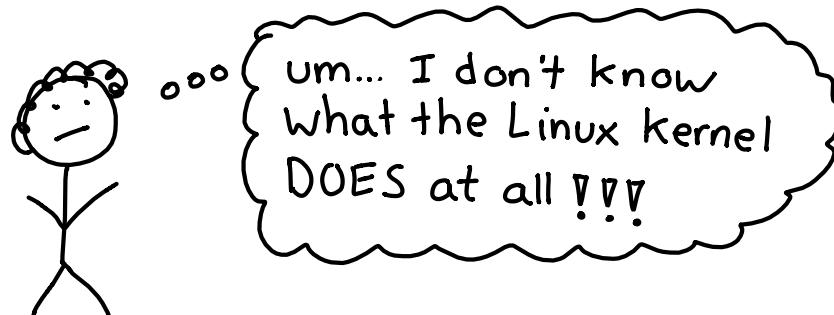
It's super fun to see a progression like:



I think a) picking something to focus on and
b) *actively* working on getting better at it is how all the people I admire got where they are

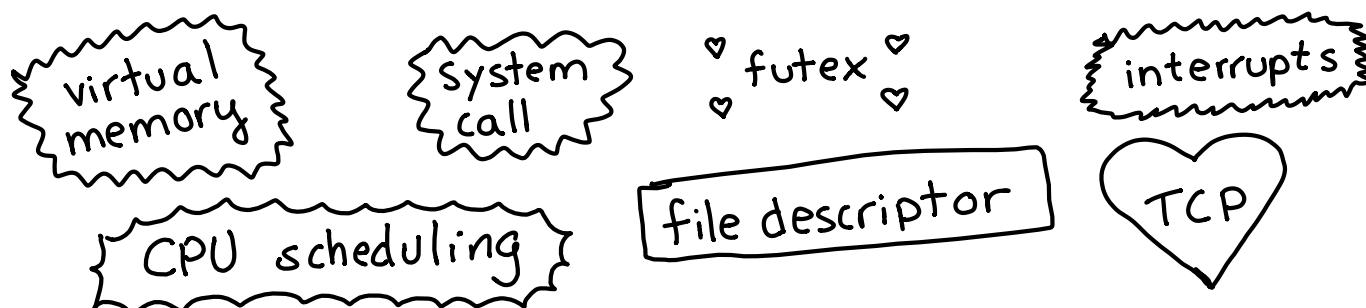
it's not too late to start learning

I started learning Linux in high school, in 2003. In 2013, after using it every day for 10 years, I realized something kind of scary:

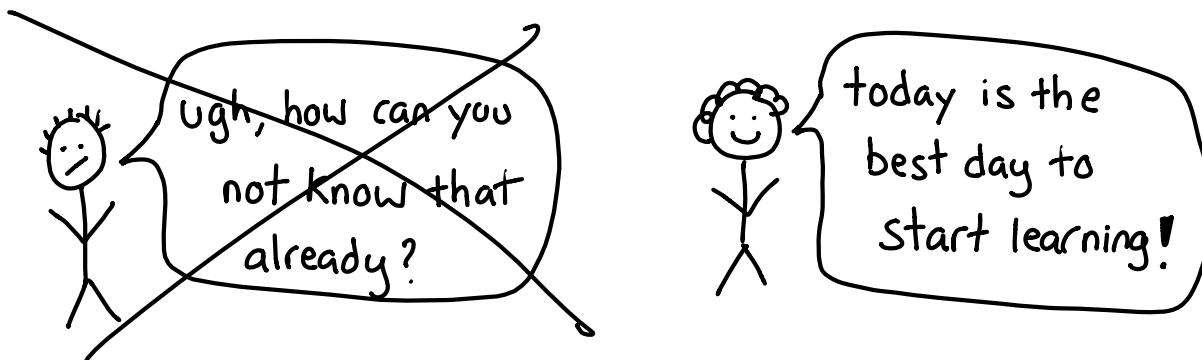


Julia, 2013

There were all KINDS of concepts that I either didn't understand or didn't even know existed:

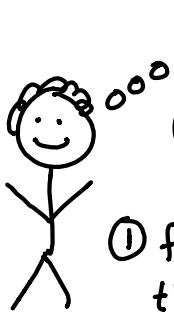


Just today (in 2017!), I realized I don't fully understand how Linux users/groups work. No big deal! I picked up my copy of "The Linux Programming Interface", read Chapter 9, and now I understand.



ways to build expertise

learn fundamental concepts



"System call"?
What's that?

- ① figure out which ideas are the most important
- ② Learn them!



do experiments!

what system calls
is THIS program using?

what's
in /proc?

What happens if I
run out of memory
on purpose?

how long
does it take
to read 5GB
from disk?

read books



even just reading a
few chapters of a
good book can help!

do hard projects



ooo ooh, I'd need to
learn a lot more
to do that
project

I'll work on
that!

When you don't understand something, dig in



... that's weird...

* figure it out *

I learned something
new!!!

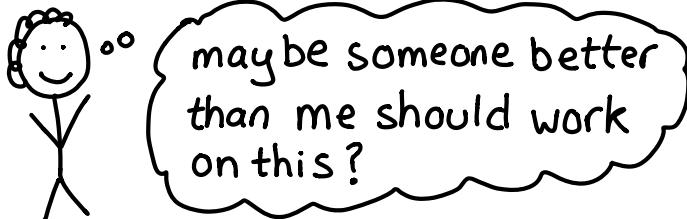
don't forget: it takes a long time



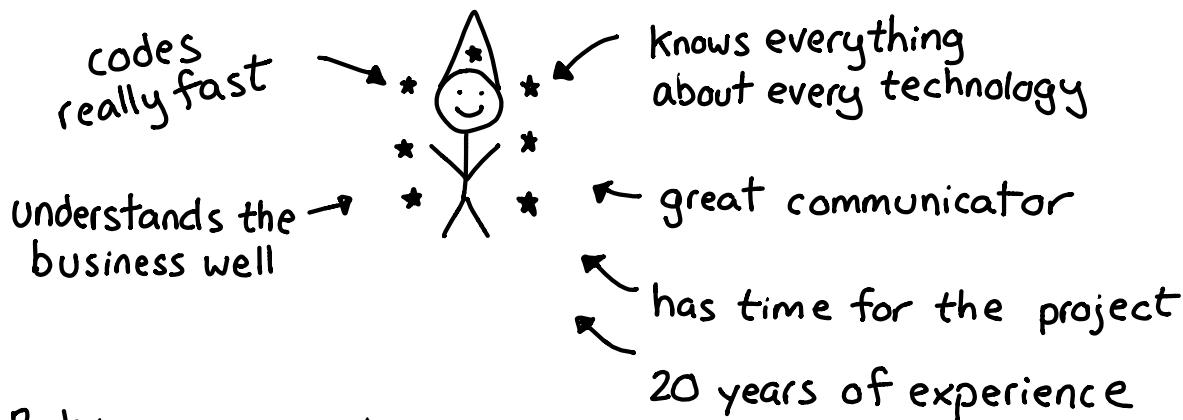
after 3 years, I know
a lot... but there's
a lot more still !!!

take on hard projects

To wrap up, let's talk about one last wizard skill: confidence.
When there's a hard project, sometimes I think:



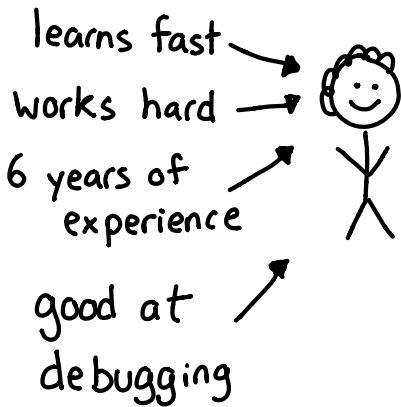
and I imagine this ★ magical ★ human:



But in programming:

- We're changing the tech we use all the time.
- every project is different, and it's rarely obvious how to do it.
- there aren't many experts, and they certainly don't have time to do everything.

So instead, we have me:



I figure "someone's gotta do this", write down a plan, and get started! A lot of the time, it turns out well. I learn something and feel a little more like a Wizard ♥ ♥

